STABLE - MATCHING - PROBLEM

- Instabile Paare:
  In Matching M heißt ein nicht zugewiesenes Paar instabil, wenn die beteiligten Partner jeweils sich gegenseitig gegenüber den aktuellen Partners bevorzugen

  -> STABLE MATCHING bezeichnet ein Matching ohne instabile Paare

- Lösungen:
  - BRUTE - FORCE: $n!$
  - GALE - SHAPLEY - ALGORITHMUS: n^2
    - PSEUDOCODE
      - mit Arrays:

```
for i <- 0 bis n-1
    for j <- 0 bis n-1
        Ranking [i, FPref [i,j]] <- j
for i <- 0 bis n-1
    Next[i] <- 0
    Current [i] <- -1
SFree <- Liste aller Kinder
while SFree nicht leer
    s <- lösche erstes Element & speichere hier aus SFree
    f <- SPref [s, Next[s]]
    s' <- Current [f]
    if (s' == -1)
        Current [f] <- s
    else if (Ranking [f, s] < Ranking [f, s']
        Current [f] <- s
        Füge s' in SFree an erster Stelle ein
    else
        Füge s in SFree an erster Stelle ein
    Next [s] <- Next[s] + 1
```

  - KORREKTHEIT:
    - Terminierung: -> alle Kinder & Familien (Partner) werden zugewiesen
      -> kann nicht sein, da das Kind alle Familien bereits gewählt hat, da sonst zugewiesen

- Stabilität: Gale-Shapley liefert immer Stable Matching
  - → entweder Partner A hat B noch nicht ausgewählt
    - ⇒ nicht instabil
  - → oder B hat A abgelehnt
    - → nicht instabil
  - ⇒ am Ende immer stabile Paare

ANALYSE VON ALGORITHMEN

- Maße für Effizienz von Algorithmen:
  - Laufzeit
  - Speicherplatz
  - Besondere charakterisierende Parameter (problemabhängig)
    z.B.: Anzahl Vergleichsoperationen oder Bewegungen von Datensätzen beim Sortieren

- Laufzeitanalyse:
  - MASCHINENMODELL
    - RAM (Random Access Machine)
      - genau ein Prozessor
      - alle Daten am Hauptspeicher
      - Speicherzugriffe dauern alle gleich lang
  - PRIMITIVE OPERATIONEN
    - Zeit für eine Operation = eine Zeiteinheit
    - Arten:
      - Zuweisungen ($a \leftarrow b$)
      - Vergleichsoperationen (if, else, ...)
      - Arithmetische Befehle ($+, -, \cdot, /, \text{mod}, \ldots$)
      - Logische Operationen ($\vee, \wedge, \neg, \ldots$)
    - dabei können Indexrechnung, Typ & Länge der Operanden vernachlässigt werden
  - LAUFZEIT
    - Anzahl an primitiven Operationen
    - Laufzeit $T(n)$ ist Funktion der Eingabegröße n (Eingabe wird als Instanz bezeichnet)
  - WORST-CASE-LAUFZEIT: größtmögliche Laufzeit
    - Effizienz in der Praxis
  - AVERAGE-CASE-LAUFZEIT: durchschn. Laufzeit über alle möglichen gültigen Eingaben

- BEST-CASE-LAUFZEIT: best mögliche Eingabe
  - untere Schranke → nur in Ausnahmefällen erreichbar

- Konventionen für Pseudocode:
  - Schlüsselwörter: fett oder in Farbe
  - Zuweisungen: ←
  - Vergleich: $<, \leq, =, \neq, >, \geq$
  - Negation: !
  - Bedingungen:
    - nicht geklammert
    - komplexe → ganze Sätze
  - Blockstruktur:
    - statt Klammern
    - mehrere Ausdrücke mit Beistrichen in einer Zeile
  - Funktionen:
    - Funktionsnamen: BFS(s): code hier
    - Arrays werden per Referenz übergeben
      → Änderungen im Arrayinhalt in Funktion auch außerhalb sichtbar
  - Speicher:
    - Freigabe wenn nicht benötigt automatisch (garbage collection)

- Asymptotisches Wachstum: Vergleich von Algorithmen unabhängig von Maschinenparametern
  - GRUNDIDEE:
    - arbeiten mit Schranken
    - keine konstanten Faktoren
    - keine kleinen n
  - OBERE SCHRANKE: $O(f(n))$ - Menge aller Funktionen, die $f(n)$ asymptotisch durch $c \cdot f(n)$ von oben beschränkt
    - Definition: $T(n)$ ist in $O(f(n))$, wenn Konstanten $c > 0$ & $n_0 > 0$ existieren, so dass $T(n) \leq c \cdot f(n) \; \forall \, n \geq n_0$
    - Schreibweisen:
      - $T(n)$ ist in $O(f(n))$
      - $T(n) \in O(f(n))$
      - $T(n) = O(f(n))$

- UNTERE SCHRANKE: $\Omega(f(n))$ ist Menge aller Funktionen, die asymptotisch durch $c \cdot f(n)$ von unten beschränkt werden
  - Definition: $T(n) = \Omega(f(n))$, wenn Konstanten $c > 0$ und $n_0 > 0$ existieren, so dass $\forall\ n \geq n_0$ $T(n) \geq c \cdot f(n)$ gilt
- SCHARFE SCHRANKE: $\Theta(f(n))$ ist Menge aller Funktion, die asymptotisch gleich großes Wachstum wie $c \cdot f(n)$ haben
  - Definition: $T(n) = \Theta(f(n))$, wenn $T(n)$ sowohl in $O(f(n))$ als auch in $\Omega(f(n))$ ist
- EIGENSCHAFTEN:
  - Additivität:
    - Obere Schranke: wenn $f = O(h)$ und $g = O(h)$ $\Rightarrow f+g = O(h)$
    - Untere Schranke: wenn $f = \Omega(h)$ oder $g = \Omega(h)$ $\Rightarrow f+g = \Omega(h)$
    - Scharfe Schranke: wenn $f = \Theta(h)$ & $g = \Theta(h)$ $\Rightarrow f+g = \Theta(h)$
  - Transitivität:
    - Obere Schranke: wenn $f = O(g)$ und $g = O(h)$ $\Rightarrow f = O(h)$
    - Analoges für untere & scharfe Schranke
  - Äquivalenz: $f = \Theta(g)$ bildet Äquivalenzrelation
    - → Symmetrie: $f = \Theta(g)$ genau dann, wenn $g = \Theta(f)$
    - → Reflexivität: $f = \Theta(f)$
    - → Transitivität
- ASYMPTOTISCHE DOMINANZ
  - f wird von g dominiert, wenn $f = O(g)$
    ! ACHTUNG: Umkehrung gilt nicht!
  - f und g nicht in gleicher Äquivalenzklasse bzgl. asymptotischem Wachstum
  - Schreibweise: $f \ll g$
  - Alternative Schreibweise: g dominiert f, wenn $\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0$
    ! Bei nicht-stetigen Funktionen ist keine Dominanz gegeben!

- $O(n \log(n))$-WACHSTUM: leicht überlinear
  - Beispiele:
    - Divide & Conquer
      -> Mergesort
    - Größtes leeres Intervall:
      -> n Zeitpunkte gegeben
      -> maximales Intervall, in dem keine neuen Daten ankommen ist gesucht
      -> Lösung:
        - Sortiere Zeitpunkte
        - Liste in sortierter Reihenfolge durchlaufen & jeweils max. Intervall berechnen

- Quadratisches Wachstum: $O(n^2)$
  - Beispiele:
    - Dichtestes Punktepaar
      -> alle Paare von Punkten überprüfen

```
min ← (x1 - x2)^2 + (y1 - y2)^2
for i ← 1 bis n-1
    for j ← i+1 bis n
        d ← (xi - xj)^2 + (yi - yj)^2
        if d < min
            min ← d
```

- Kubisches Wachstum: $O(n^3)$
  - Beispiele:
    - Disjunkte Mengen?
      n Mengen $S_1 \dots S_n$ wobei jede Menge Teilmenge von $\{1, 2, \dots, n\}$ ist
      Existiert Paar von Mengen das disjunkt (kein gemeinsames Element)
      -> für jedes Paar von Mengen schauen ob d.

```
for i ← 1 bis n-1
  for j ← i+1 bis n
      foreach Element p von Si
          p in Sj vorhanden?
      if kein Element aus Si in Sj
          gib aus dass Si & Sj disjunkt
```

- POLYNOMIELLES WACHSTUM: $O(n^k)$
  - Beispiel: Teilsummenproblem mit k-Zahlen existiert Teilsumme, so dass Summe einer vorgegebenen Zahl entspricht

    k = 4 (Teilsummengröße) - ineffizient

```
for i ← 0 bis n-4
    for j ← i+1 bis n-3
        for k ← j+1 bis n-2
            for l ← k+1 bis n-1
                if (A[i] + A[j] + A[k] + A[l]) = X
                    Teilsumme gefunden
```

    Effizienter: Wahr/Falsch-Matrix befüllen

- EXPO-WACHSTUM: $O(c^n)$
  - Beispiel: Teilsummenproblem mit allen Untermengengrößen

```
foreach Teilmenge S von Zahlen
    überprüfe ob Teilmengensumme gleich x
```

    → endliche Menge mit n Elementen hat 2^n Teilmengen; jede muss untersucht werden

- Einfache Datenstrukturen:
  - ARRAY: statische Datenstruktur mit im Speicher sequentiellen Elementen
    - alle Elemente gleicher Typ
    - Zugriff mit Index in konstanter Zeit
    - Zweidimensionale Arrays: "Array von Arrays"
  - VERKETTETE LISTE: dynamische Datenstruktur
    - Speicherung Knoten, die miteinander in Beziehung stehen
    - Anzahl Knoten muss im Vorhinein nicht bekannt sein
  - VERGLEICH ARRAY & LISTE

	A	L
Element suchen	$\Theta(n)$	$\Theta(n)$
auf Element i zugreifen	$\Theta(1)$	$\Theta(n)$
Element am Anfang einfügen	$\Theta(n)$	$\Theta(1)$

} worst Case

- Sortieren:
  - offensichtliche Anwendungen: Namenslisten sortieren, Google-Resultate ordnen, ...
  - Vereinfachg anderer Probleme: binäre Suche, Median finden, ...

- ELEMENTARE SORTIERVERFAHREN:
  - Bubblesort: Worst / Average: $O(n^2)$
    Best: $O(n)$
    - Funktion: Vergleich ob nächstes Element kleiner
      → falls ja tauschen
      → sonst Vergleich mit nächstem Element durchführen
  - Selectionsort (Minimumsuche): immer $O(n^2)$
    - Funktion: kleinstes Element aus unsortiert an Ende von sortiert anfügen

```
Selection Sort (A):
  for i ← 0  bis  n-2
      smallest ← i
      for j ← i+1  bis  n-1
          if A[j] < A[smallest]
              smallest ← j
      Vertausche A[i] mit A[smallest]
```

    - Vertauschungen: $\Theta(n)$
  - Insertionsort (Einfügen): Worst: $\Theta(n^2)$
    Average: $\Theta(n^2)$ → $\frac{i}{2}$-mal
    Best: $\Theta(n)$
    - Funktion: Element an richtiger Stelle einfügen

```
Insertion Sort (A):
  for i ← 1  bis  n-1
      key ← A[i], j ← i-1
      while j ≥ 0 und A[j] > key
          A[j+1] ← A[j]
          j ← j-1
      A[j+1] ← key
```

    - Vertauschungen: $\Theta(n)$

- Polynomialzeit:
  $\exists$ eine Konstante $d \geq 1$, so dass Laufzeit in $O(n^d)$

  → Brute-Force meist nicht polynomiell & daher zu zeitaufwendig

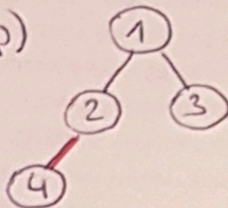
- Definition: Algorithmus ist effizient, wenn Laufzeit polynomiell

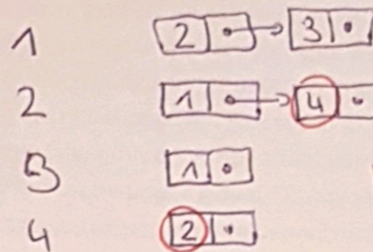
GRAPHEN

- Anwendungen: U-Bahn-Pläne, Karten, Webgraphen, Verkehr, ..
- Ungerichtete / gerichtete Graphen:
  - UNGERICHTET: $G = (V, E)$
    - V = Menge Knoten (vertices, nodes)
    - E = Menge Kanten (edges)
      → Kante zw. a und b → (a,b) oder (b,a) bzw a − b bzw. b − a
    - Anzahl Knoten $= n = |V|$
    - Anzahl Kanten $= m = |E|$
    - $(a,b) = (b,a)$
    - Adjazent: falls Knoten u & v durch Kante verbunden sind; u & v sind Nachbarn
    - Inzident: $e = (u,v) \Rightarrow v/u$ und e sind inzident
    - Knotengrad (degree): $deg(u)$
      → Anzahl zu u inzidente Kanten
      → Handschlaglemma: $\sum_{u \in V} deg(u) = 2 \cdot |E|$
    - Schlichter Graph: keine Mehrfachkanten oder Schleifen
  - GERICHTET: Digraph $\Rightarrow G = (V, E)$
    → wobei E Menge gerichteter Kanten (arcs)
    → Notation Kante: (a,b) oder $a \to b$
    - $a \to b \neq b \to a$
    - Eingangsknotengrad $deg^-(v)$
    - Ausgangsknotengrad $deg^+(v)$
      → $deg(v) = deg^+(v) + deg^-(v)$

- Adjazenzmatrix: repräsentiert Graphen
  → n×n - Matrix mit $A_{uv} = 1$, wenn (u,v) eine Kante besitzt
  - 2 Einträge für jede ungerichtete Kante
  - Gewichtete Graphen ⇒ Reelle Matrix statt "Boolsche"
  - Platzbedarf: $\Theta(n^2)$ (daher auch Ausgabe in $\Theta(n^2)$)
  - Überprüfung ob Kante: $\Theta(1)$ - konstant
- Adjazenzlisten: repräsentiert Graphen
  → Array von Listen wobei der Index die Knotennr. ist
    → 2 Einträge für jede Kante
  - bei gewichteten: Gewicht in Liste speichern
  - Platzbedarf: $\Theta(m+n)$ → Aufzählen aller Kanten ebenfalls
  - Überprüfung ob Kante: $O(\deg(u))$

Bsp)





Kante zweimal

- Adjazenzmatrix vs. Adjazenzlisten:
  → Graph kann bis zu $m = \frac{n(n-1)}{2} = \binom{n}{2} = \Theta(n^2)$ viele Kanten haben
  → Graphen sind dicht wenn $m = \Theta(n^2)$ (dense)
  und licht wenn $m = O(n)$ (spare)
  → für dichte Graphen beide Darstellungen vergleichbar
    - jedoch in der Praxis hauptsächlich lichte Graphen
      ⇒ Liste verwenden

! Falls Algorithmus auf Graph in Linear Zeit läuft ⇒ Darstellung in Adjazenzliste & Laufzeit von $O(n+m)$!

- AUSGABE VON KANTEN
  - Matrix - Gerichteter Graph

```
for u ← 0 bis n-1
    for v ← 0 bis n-1
        if M[u,v] = 1
            Ausgabe
```

n Knoten (0 - n-1)
Matrix M

- Liste – Gerichteter Graph

```
for u ← 0 bis n−1
   foreach Kante (u,v) inzident zu u
      Ausgabe
```

- Matrix – Ungerichteter Graph

```
for u ← 0 bis n−2          ← sofern Graph schleiff bleibt Diagonale leer
   for v ← u+1 bis n−1
   if M[u,v] = 1
      Ausgabe
```

- Liste – Ungerichteter Graph

```
for u ← 0 bis n−1
   foreach Kante (u,v) inzident zu u
      if u<v
         Ausgabe
```

- Pfade und Zusammenhang
  - PFAD
    - → in ungerichtetem Graph $G = (V, E)$ ist Knotenfolge $v_1, v_2, \dots, v_k$ mit $k \geq 1$ Pfad, wenn alle aufeinanderfolgenden Paare v_i, v_{i+1} adjazent
      - Länge: $k-1$
    - → Erreichbarkeit: u ist von v erreichbar, falls G u-v-Pfad enthält (Pfad geht von u nach v)
  - ZUSAMMENHANG
    - → zusammenhängender Graph: jedes Paar von Knoten ist über Pfad erreichbar
  - DISTANZ:
    - → zw. Knoten u&v ist Länge kürzester u-v-Pfad (= einfach d.h. jeder Knoten nur einmal)

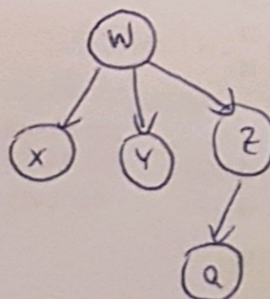
! Falls u von v nicht erreichbar → Distanz $= \infty$!

- Kreise:
  → Pfad $v_1, v_2, \dots, v_k$ in dem $v_1 = v_k$ und $k \geq 4$ & die ersten $k-1$ Knoten unterschiedlich
  - Länge: $k-1$
- Pfade & Kreise in gerichteten Graphen:
  → selbe Definition für Pfad, nur dass Kanten gerichtet
  → Kreis selbe Definition, nur, dass $k \geq 3$ reicht

BÄUME

- Definition: ungerichteter Graph, der kreisfrei & zusammenhängend
  → G sei ungerichteter Graph mit n Knoten
  ⇒ je 2 Aussagen implizieren dritte:
  - G zusammenhängend
  - G enthält $n-1$ Kanten
  - G ist kreisfrei

  → G sei ungerichteter Graph
  ⇒ G genau dann Baum, wenn für jedes Paar von Knoten genau ein Pfad existiert (sonst nicht kreisfrei (mehrere) oder zusammenhängend)
- Wurzelbaum: modelliert hierarchische Strukturen
  → jede Kante ausgehend vom Wurzelknoten w hat Kante mit Richtung weg von w



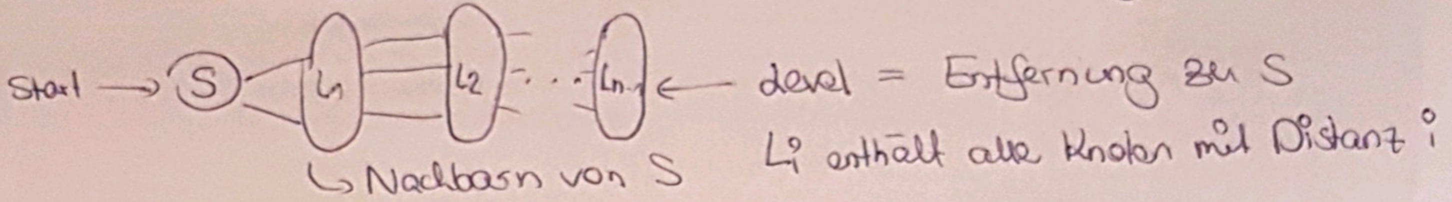
rooted tree / arborescence

DURCHMUSTERUNG VON GRAPHEN (Graph Traversal)

• Breitensuche (Breadth First Search, BFS):

→ Abarbeitung Ebene für Ebene

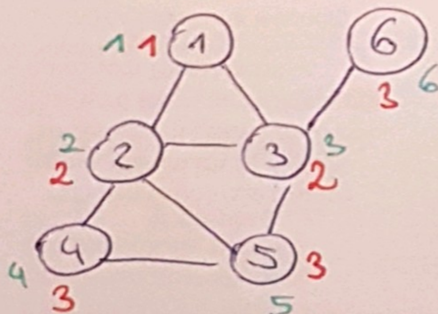
→ von Startknoten aus in alle Richtungen suchen



• Anwendungen: Facebook, Labyrinth, kleinste Anzahl von Hops

```
BFS(G, s):
Discovered [s] ← true
Discovered [v] ← false  ∀ anderen Knoten v ∈ V
Q ← {s}          // Queue
while Q nicht leer
      ersten Knoten aus Q entfernen  = u
      // hier Aktion auf u ausführen
      foreach Kante (u, v) inzident zu u
            if !Discovered [v]
                  Discovered [v] ← true
                  v zu Q hinzu
```

Bsp)



Level

mögliche Reihenfolge

• Laufzeit: $O(m+n) \rightarrow m = 2m = \sum_{u \in V} deg(u)$

• BFS-Baum: Breitensuche erzeugt Baum → alle von S erreichbaren Knoten

→ Kanten in Baum können sich um max. eine Ebene unterscheiden

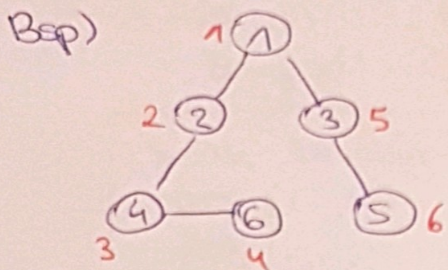
• Tiefensuche (Depth First Search, DFS):
→ von besuchtem Knoten u immer zuerst zu untersuchten bevor andere Nachbarn von u betrachtet

```
DFS(G,s):
Discovered[v] ← false ∀ Knoten v ∈ V
DFS1(G,s)

DFS1(G,u):
Discovered[u] ← true
// hier Aktionen auf u ausführen
foreach Kante (u,v) inzident zu u
    if !Discovered[v]
        DFS1(G,v)
```

falls keine Nachbarknoten mehr → unterste Rekursionsstufe verlassen

Bsp)



Reihenfolge

• Laufzeit: $O(m+n)$ → $m = 2m = \sum_{u \in V} deg(u)$

ZUSAMMENHANGSKOMPONENTEN

• Zusammenhangskomponente in ungerichtetem Graph:
→ maximal zusammenhängender Teilgraph ($= G_1 = (V_1, E_1)$; enthält Teilmenge von Knoten & Kanten von G)
⇒ nicht zusammenhängender Graph besteht aus einzelnen Zusammenhangskomponenten
→ können mit BFS oder DFS bestimmt werden (nur eine davon ⇒ da nicht zusammenhängend)

• Zusammenhangskomponenten zählen:

```
DFSNUM(G):
Discovered[v] ← false ∀ Knoten v ∈ V
i ← 0        // Anzahl
foreach Knoten v ∈ V
    if Discovered[v] = false
        i ← i+1
        DFS1(G,v)
return i
```

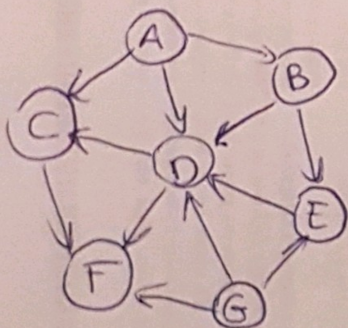
da Discovered global → DFS1 setzt alle in Zusammenhangsk. auf true → nur Komponenten gezählt

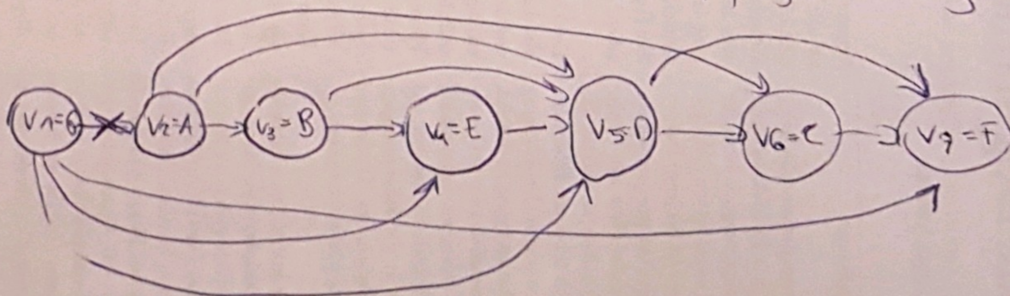
Laufzeit: $O(n+m)$

- **Zusammenhang in gerichteten Graphen:**
  - DEFINITION: u & v sind gegenseitig erreichbar, wenn Pfad von u nach v & von v nach u existiert
    - -> wenn jedes Knotenpaar gegenseitig erreichbar => STARK ZUSAMMENHÄNGEND
    - -> bei missachten der Richtg gegenseitig erreichbar (= nur erreichbar -> ungerichteter G) => SCHWACH ZUSAMMENHÄNGEND
  - LEMMA: s sei beliebiger Knoten in gerichtetem G => G stark zusammenhängend dann und nur dann, wenn jeder Knoten von s aus und s von jedem Knoten aus erreicht werden kann
  - ALGORITHMUS ZUR ÜBERPRÜFUNG OB STARK Z.
    - Laufzeit: $O(m+n)$

```
wähle beliebigen Knoten s
BFS (G, s)
BFS (Grev, s)   // umgedrehte Orientierg der Kanten in G
for alle Knoten
if BFS1[Knoten] && BFS2[Knoten]   // Boolesche Arrays
   STARK ZUSAMMENHÄNGEND
```

- **Gerichteter azyklischer Graph (Directed Acyclic Graph, DAG):**
  - DEFINITION: DAG = gerichteter Graph ohne gerichtete Kreise
  - QUELLE: Knoten v ohne eingehende Kante $\hat{=} \deg^-(v)=0$
  - TOPOLOGISCHE SORTIERUNG: von gerichtetem Graphen
    - -> lineare Ordnung seiner Knoten mit $v_1, v_2, \dots, v_n$ sodass jede Kante (v_i, v_j) $i<j$ erfüllt





- REIHENFOLGEBESCHRÄNKUNG
  Kante(u, v) bedeutet, dass Aufgabe u vor v
- LEMMAS:
  - Wenn G topologische Sortierung hat ⟺ G ist DAG
  - wenn G ein DAG besitzt G eine Quelle
  - G genau dann DAG, wenn jeder Teilgraph Quelle besitzt
- ERKENNEN DAG: jeder Knoten wird irgendwann zur Quelle (außer letzter)

```
while G hat mindestens einen Knoten
  if G hat Quelle
    wähle eine Quelle v aus
     v Ausgabe
    Lösche v & alle inzidenten Kanten aus G
  else return G kein DAG
return G ist DAG
```

- ERSTELLUNG TOPOLOGISCHE SORTIERUNG

```
foreach v ∈ V
   count[v] ← 0
foreach v ∈ V
   foreach Kante (v, w) ∈ E
      count[w] ← count[w] + 1
foreach v ∈ V
   if count[v] = 0
      v zur Liste am Anfang hinzu
while L ist nicht leer
    Sei v erstes Element in L, lösche v aus L
    Gib v aus
    foreach Kante (v, w) ∈ E
        count[w] ← count[w] - 1
        if count[w] = 0
            Gib w zur Liste L an Anfang hinzu
```

- Laufzeit: $O(n+m)$

- Kürzester Pfad (Shortest Path Problem)
  - DIJKSTA:
    - mit Liste:

```
Dijkstra(G,s):
Discovered[v] ← false ∀ Knoten v ∈ V
d[s] ← 0
d[v] ← ∞  ∀ anderenen Knoten v ∈ V \ {s}
L ← V
while L nicht leer
    wähle u ∈ L mit kleinstem Wert d[u]
    lösche u aus L
    Discovered[u] ← true
    foreach Kante e = (u,v) ∈ E
        if(!Discovered[v])
            d[v] ← min(d[v], d[u] + l_e)
                                  ↑ Länge Kante
```

→ Laufzeit: $O(n^2)$ Worst-Case

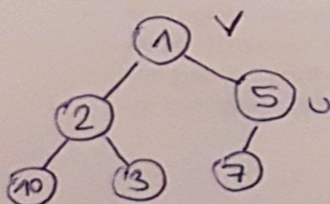
- mit Priority Queue:

Priority Queue:
  → Datenstruktur, die Menge S von Elementen verwaltet
  → jedes Element $v \in S$ hat dazugehörigen Wert i (= Priorität)
    - je kleiner i desto höher Prio
  → Einfügen, Löschen & Element mit kleinstem Wert finden in $O(\log(n))$
  → nutzt Heap

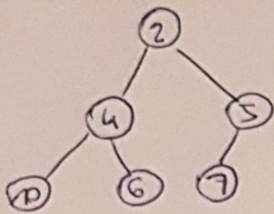
Heap: Heap (Min-Heap) ist binärer Wurzelbaum dessen Knoten total geordnet sind so dass:
- u linkes o. rechtes Kind von v → wenn $v \leq u$
- letzte Ebene wird linksbündig aufgefüllt

! MAX-HEAP: !
$u \leq v$

Bsp)



Min-Heap



Index	x	1	2	3	4	5	6
Wert	x	2	4	5	10	6	7

- Repräsentation Heap:
  - -> Knoten von Baum in Array speichern wobei Index 0 leer bleibt
  - => Berechnung effizienter
    - -> linke Nachfolger: $2k$
    - rechts: $2k+1$
    - -> Elternknoten: $\lfloor \frac{k}{2} \rfloor$

- Heapify-up : $O(\log(n))$
  - -> durch Einfügen an der Stelle $n+1$ kann Heapbedingung verletzt werden

tauscht Element solange nach oben, dass Heapbedingung wieder korrekt ->

```
Heapify-up (H,i):
if i > 1
  j <- ⌊i/2⌋
  if H[i] < H[j]
    Vertausche H[i] & H[j]
    Heapify-up(H,j)
```

- Heapify-down: $O(\log(n))$
  - -> durch Löschen & verschieben von letzter Stelle an die freie Stelle kann Heapbedingung verletzt werden
    - -> Element zu groß -> heapifydown
    - -> zu klein -> Heapifyup

| Da normalerweise nur Wurzel entfernt -> nur heapifydown. |

```
Heapify-down (H,i):
n <- length(H) - 1
if 2·i > n
  return
elif 2i < n
  left <- 2i , right <- 2i+1
  j <- Index des kleineren Wertes von H[left]
       und H[right]
else
  j <- 2i
if H[j] < H[i]
  vertausche H[i] & H[j]
  Heapifydown(H,j)
```

- Heap-Operationen
  - → Insert (H, v)
    $O(\log(n))$
  - → FinMin(H) / FindMax(H): konstant je nach Min-oder Maxheap
  - → Delete(H, i)
    $O(\log(n))$
  - → ExtractMin: Min löschen
    $O(\log(n))$
- Erstellen Heap: $O(n)$

```
Init (A, n):
for i = ⌊n/2⌋ bis 1
    Heapify-down (A, i)
```

→ Dijkstra:

```
Dijkstra (G, s):
Discovered [v] ← false   ∀ Knoten v ∈ V
d[s] ← 0
d[v] ← ∞   ∀ anderen Knoten
Q ← V
while Q nicht leer
    wähle u ∈ Q mit kleinstem Wert d[u]
    lösche u aus Q       // heapify-down
    Discovered [u] ← true
    foreach Kante e = (u, v) ∈ E
        if !Discovered [v]
            if d[v] > d[u] + l_e
                lösche v aus Q   // heapify-down
                d[v] ← d[u] + l_e
                füge v Q hinzu   // heapify-up
```

→ Laufzeit: $O((n+m)\log(n))$

ALGORITHMEN - PARADIGMEN

- Greedy: inkrementelle Lösungserstellung durch Wahl lokales Kriterium zum Finden nächster Lösungskomponente
- Devide-and-Conquer: Probleme in Teilprobleme, die unabhängig voneinander gelöscht werden können
  → Kombi Teillösungen zu Gesamtlösung

GREEDY - ALGORITHMEN

→ Lösung schrittweise aufgebaut → in jedem Schritt wird Problem auf kleineres reduziert
→ Greedy-Prinzip: wähle jeweils beste (lokales Kriterium) Lösungskomponente
→ Entscheidungen werden nicht zurückgenommen
→ kann optimale Lösung liefern muss aber nicht

- Beweis optimale Lösung (falls vorhanden)
  - Ausgang: optimale Lösung wird nicht geliefert
  - heißt es gibt bessere Lösung
  - die zB: mehr Geld bringt, früher fertig ist
  - Optimalere Lösung enthält Element, das durch letztes Greedy-Element ersetzbar ⇒ ↯

- Intervall Scheduling:
  - Mögliche Greedy-Strategien:
    - Früheste Startzeit - Gegenbeispiel
    - Früheste Beendigungszeit
    - Kleinstes Intervall - Gegenbeispiel
    - Wenigste Konflikte - Gegenbeispiel

  → aufsteigende Reihenfolge Beendigungszeit liefert optimales Ergebnis

Laufzeit: $O(n \log(n))$ ↑ Sortierung
Greedy selbst: $O(n)$

```
Sortiere Jobs nach Beendigungszeit, sodass f1 ≤ f2 ≤ ... ≤ fn
A ← ∅
t ← 0
for j ← 1 bis n
    if t ≤ sj
        A ← A ∪ {j}
        t ← fj
return A
```

- **Minimaler Spannbaum**: Minimum Spanning Tree, MST
  = Teilgraph $G_T = (V, T)$ von G mit gleicher Knotenmenge & $T \subseteq E$, sodass Summe der Kantengewichte minimal

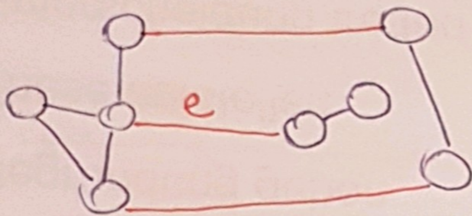
  G = zusammenhängend, schlicht mit Kantengewichten $\in \mathbb{R}$

  → Brute-Force-Ansatz nicht effizient, da exponentiell viele Spannbäume

  - Anwendungen:
    - Netzwerkentwurf (Telefonie, Computernetze, ...)
    - Approximation schwerer Probleme (Travelling Salesman, ...)

  - LEMMATA:
    - Kantenschnittlemma: - S sei beliebige Teilmenge von Knoten
      - e sei minimalgewichtete Kante mit genau einem Endknoten S

      ⇒ MST enthält e



      - Kantenschnittmenge: S Teilmenge der Knoten. D = Kantenschnittm. enthält alle Kanten mit genau einem Endpunkt in S

    - Kreislemma: - C sei beliebiger Kreis
      - f maximalgewichtete Kante

      ⇒ MST enthält f nicht (da sonst nicht mehr Baum)

    - Paritätslemma:
      ein beliebiger Kreis & eine beliebige Kantenschnittmenge haben gerade Anzahl von Kanten gemeinsam

  - ALGORITHMUS VON PRIM:

```
Prim (G, c):
foreach (v∈V)
    A[v] ← ∞
Initialisiere leere Priority Queue Q
foreach (v∈V)
    Füge v in Q ein
S ← ∅          // Kantenschnittmenge bzw. Teilmenge
while Q nicht leer:
    u ← minimales Element aus Q entnehmen
    S ← S ∪ {u}
```

Kantenschnittlemma anwenden (Randnotiz zu den letzten Zeilen)

alle Nachbarn anschauen {

```
foreach Kante e = (u, v) inzident zu u
    if (v ∉ S und c_e < A[v])
        Verringere Priorität A[v] auf c_e
```

- Laufzeit:
  - mit Priority Queue in Array: $O(n^2)$
  - mit MinHeap als Priority Queue: $O(m \log(n))$

– ALGORITHMUS VON KRUSKAL:

```
Kruskal (G, c):
Sortiere Kantengewichte so, dass c1 ≤ c2 ≤ ... ≤ cm
T ← ∅
foreach (u ∈ V) erzeuge einelementige Menge mit u
for i ← 1 bis m
    (u, v) = ei
    if u und v in verschiedenen Mengen          ← Kreislemma
    T ← T ∪ {ei}                                ← Kantenschnittlemma
    Vereinige Mengen mit u & v
return T
```

→ Kreislemma: sind u & v in versch. Zusammenhangsk.
- TFS oder BFS
- effizienter: Union – Find
  → Sortierung: $O(m \log(n))$
  bzw $m \leq n^2 \Rightarrow \log m$ in $O(\log(n))$

→ Kantenschnittlemma: union für Vereinigung

- Union-Find-Datenstruktur: abstrakter Datentyp
  → dynamische disjunkte Mengen (DDM)
  → Familie $S = \{S_1, S_2, \dots, S_k\}$ disjunkter Teilmengen von M; S_i hat Repräsentant
  - Makeset: erzeugt Menge $\{v\} = S_v$; v Repräsentant
    parent[v] = v — Laufzeit: $O(n)$
  - Union: vereinigt S_v & S_w; neuer R. beliebig
    parent[v] = w — Laufzeit: $O(n)$

- findset: liefert Repräsentant der Menge S

```
h = v
while parent[h] ≠ h
   h = parent[h]
return h
```

Laufzeit: $O(n)$

- VERGLEICH DER ALGORITHMEN:
  - Kruskal: Gesamtaufwand durch Kantensortieren
    → $O(m \log(n))$
  - Prim: – mit Priority Queue & klassischem Heap
    → $O(m \log(n))$
    – Fibonacci-Heap
    → $O(m + n \log(n))$

  → in Praxis:
  - Prim besser für dichte Graphen ($m = \Theta(n^2)$)
  - Kruskal besser für dünne Graphen ($m = \Theta(n)$)

DEVIDE-AND-CONQUER

→ teile Problem in mehrere Teile auf (meist 2)
→ teile rekursiv lösen
→ Subproblemlösungen zu Gesamtlösung verbinden

- Mergesort (Sortieren durch Mischen):

```
Mergesort (A, l, r):
if l < r
   m ← ⌊(l+r)/2⌋
   Mergesort (A, l, m)
   Mergesort (A, m+1, r)
   Merge (A, l, m, r)
```

→ Verschmelzen effizient, indem immer die ersten Elemente beider Seiten betrachtet werden & kleineres Element zuerst in Endarray übernommen wird → $O(n)$

Merge (A, l, m, r):

```
i ← l , j ← m, k ← l
while i ≤ m und j ≤ r
    if A[i] ≤ A[j]
        B[k] ← A[i], i ← i+1
    else
        B[k] ← A[j], j ← j+1
    k ← k+1
if i > m
    for h ← j bis r
        B[k] ← A[h], k ← k+1
else
    for h ← i bis m
        B[k] ← A[h], k ← k+1
for h ← l bis r
    A[h] ← B[h]
```

} i ≤ m ... linke Hälfte
j ≤ r ... rechte Hälfte
if → erstes Element links kleiner als rechts?
→ k erhält Wert
i erhöhen, k++
sonst B[k] rechter Wert, j++, k++

} *

} sonst werden mehr rechte Elemente eing. (j ≤ r nicht mehr wahr) → restliche linke hinzu

} Hilfsarray in "sortiertes" Array zurück

* falls i größer als m geworden ist (linke Hälfte komplett eingefügt)
→ nur mehr die rechte Hälfte hinzu

- ANZAHL SCHLÜSSELVERGLEICHE: $C(n)$

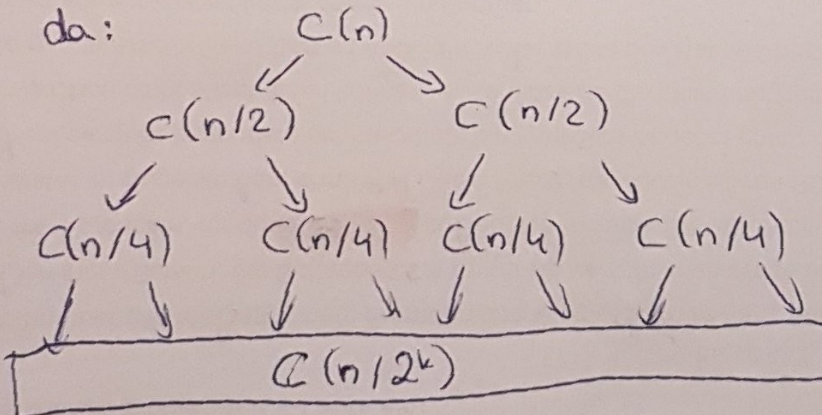
→ Mergesort Rekursionen: $C(n) \leq \begin{cases} 0 & \text{wenn } n=1 \\ C(\lceil n/2 \rceil) + C(\lfloor n/2 \rfloor) + n & \text{sonst} \end{cases}$

↙ middle ist bei links

↗ Verschmelzen

$\Rightarrow C(n) = O(n \log(n))$

da:



→ hat logarithmische Tiefe
→ in jedem Aufruf bis zu n Vergleiche pro Ebene
$\Rightarrow O(n \log(n))$

weiterer Beweis: Auflösen der Rekursion

$$\frac{C(n)}{n} \leq \frac{2C(n/2)}{n} + 1$$ ← plus 1

↗ Start ↘ erste Rekursion (beide Hälften)

→ für weiteren Rekursionen wiederholen

$$\frac{C(n)}{n} \leq \frac{2C(n/2)}{n} = \frac{C(n/2)}{n/2} + 1$$

$$\leq \frac{2C(n/4)}{n/2} + \frac{n/2}{n/2} = \frac{4C(n/4)}{n} + \frac{n/2}{n/2} + 1$$

$$= \frac{C(n/4)}{n/4} + 1 + 1$$

$$\leq \ldots$$

$$\leq \frac{C(n/n)}{n/n} + \underbrace{1 + \ldots + 1}_{\log(n)} = \log(n)$$

↗ $C(n/n) = C(1) = 0$

→ Bruch wird 0

$$\rightarrow \frac{C(n)}{n} \leq \log(n)$$

$$C(n) \leq \log(n) \cdot n$$

Beweis durch Induktion:

- Behauptung: falls $C(n)$ Rekursion erfüllt $\Rightarrow C(n) \leq n \log(n)$
  - Induktionsanfang: $n = 1$
  - Induktionsbehauptung: $C(n) \leq n \cdot \log_2(n)$
    - → Ziel: zeige dass $C(2n) \leq 2n \log_2(2n)$

$$C(2n) \leq 2C(n) + 2n$$

2n herausheben

$$\leq 2n\log_2(n) + 2n \stackrel{\downarrow}{=} 2n(\log_2(n) + 1)$$

$$= 2n\left(\log_2\left(\frac{2n}{2}\right) + 1\right)$$

$$= 2n(\log_2(2n) - \log_2(2) + 1)$$

$$= 2n(\log_2(2n) - 1 + 1)$$

$$= 2n(\log(2n))$$

⇒ Anzahl Vergleiche dominiert Laufzeit von Mergesort ∀

- **Quicksort:**
  - → wählt Pivotelement aus Folge (erstes, letztes, mittleres, zufällig = macht worst case unwahrscheinlicher, Median (erstes, letztes, mittleres))
  - → Folge in zwei Teile geteilt
    - A_1 enthält nur Elemente kleiner pivot
    - A_2 nur Elemente größer
  - → Rekursive Aufrufe A_1 & A_2
  - → einzel As wieder zusammenfügen
  - ANAYSE:
    - Best Case: - beide Teilfolgen haben fast gleiche Länge
      - Höhe Baum $\Theta(\log(n))$ (AufrufeRekursion)
      - Auf jeder Rekursionsebene $\Theta(n)$ Vergleiche
      - Vergleichsanzahl = Laufzeit: $\Theta(n \log(n))$
    - Worst Case: - jede Teilfolge wird immer beim ersten / letzten Element geteilt
      - ⇒ Anzahl Vergleiche $\Theta(n^2)$
      - Beispielsweise bei: Folge aufsteigend sortiert & Pivot ist letztes Element
        - → Rekursion nur links
        - → Teilfolge wird nur um 1 kleiner
        - → Laufzeit $\Theta(n^2)$ & Speicherplatz $\Theta(n)$

  → kann durch Wahl Pivot vermieden werden!

- **Speicherplatzkomplexität:** Maß für Anwachsen Speicherbedarf abhängig von Eingabegröße
  - Quicksort: Worst-Case: $\Theta(n)$, Best / Average: $\Theta(\log(n))$
  - Mergesort: Best / Average / Worst $\Theta(n)$
  - → Quicksort oft bevorzugt
  - VERGLEICH SORTIERVERFAHREN:

	Laufzeit / Vergleiche			Speicher		
Insertion	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
Selection	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
Quick	$\Theta(n\log(n))$	$\Theta(n \log(n))$	$\Theta(n \log(n))$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
Merge	$\Theta(n \log(n))$	$\Theta(n \log(n))$	$\Theta(n^2)$	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(n)$
	Best	Average	Worst	Best	Average	Worst

- Einsatz Sortierverfahren:
  - Quicksort: in allgemeinen Sortiersituationen bevorzugt
  - Mergesort: Sortieren von Listen & auf externen Speichern (Bottom-up-Mergesort)

→ untere Schranke für allgemeine Sortierverfahren mindestens $\Omega(n \log(n))$ im Worst Case
⇒ Mergesort ist asymptotisch zeitoptimal

⇒ • lineare Sortierverfahren: nutzen Info über Art der Werte & Wertebereich

- COUNTSORT:
  Eingabe: n Zahlen im Bereich 0 bis z in Array A; Hilfsarray Counts; $z < n$

```
for j ← 0 bis z                              } in Counts alles mit 0 belegen
    Counts[j] ← 0
for i ← 0 bis n-1                            } zählen wie oft eine Zahl vorkommt
    Counts[A[i]] ← Counts[A[i]] + 1
i ← 0
for j ← 0 bis z                              } für alle möglichen Zahlen die jeweilige
    for k ← 0 bis Count[j] - 1                 Zahl Count[Zahl] mal viele Male in A einfügen
        A[i] ← j
        i ← i + 1
```

  - Laufzeit: $\Theta(n)$

- Inversionen zählen:
  = INVERSION: i und j sind invertiert, wenn $i < j$ aber $a_i > a_j$
  - ANWENDUNG:
    - filtern ähnlicher Präferenzen
    - Aggregieren von Rängen (Metasuche im Web) → minimale Anzahl in Rangfolge finden
    - Tau-Test

- IMPLEMENTIERUNGEN:
  - Brute-Force: $\theta(n^2)$
    alle Paare vergleichen
  - Devide-and-Conquer:
    → Mergesort, der Inversionen zählt
    → Laufzeit: $O(n \log(n))$

```
Sort-and-Count (L):
if Liste L hat genau ein Element
    return 0 & Liste L
Unterteile Liste in 2 Hälften
(r_A, A) ← Sort-and-Count (A)      } Einzelinversionen
(r_B, B) ← Sort-and-Count (B)
(r, L)   ← Merge-and-Count (A, B)  } Inversionen wenn zusammen
return  r_A + r_B + r  + sortierte Liste L
```

```
Merge-and-Count (A, B):
i ← 1 , j ← 1
count ← 0
while beide Listen nicht leer
    if a_i ≤ b_j
        a_i zu Ergebnisliste ; i++
    else
        b_j zu Ergebnisliste ; j++
        count ← count + Anzahl restliche Elemente in A
Rest der nicht leeren Liste zu Ergebnisliste
return  count + Liste
```

DICHTESTES PUNKTEPAAR

- **Brute-Force-Ansatz**: alle Paare von Punkten vergleichen $\Theta(n^2)$
- **Verbesserung**: Devide-and-conquer
  → Teile: 1/2 n - Punkte auf jeder Seite
  → Herrsche: dichtestes Punktepaar rekursiv auf jeder Seite finden
  → Kombiniere: dichtestes Punktepaar über Grenze hinweg
  → Ergebnis: beste Lösung davon